Efficient Data Structures for Tamper-Evident Logging

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

Many real-world applications wish to collect tamper-evident logs for forensic purposes. This paper considers the case of an untrusted logger, serving a number of clients who wish to store their events in the log, and kept honest by a number of auditors who will challenge the logger to prove its correct behavior. We propose semantics of tamper-evident logs in terms of this auditing process. The logger must be able to prove that individual logged events are still present, and that the log, as seen now, is consistent with how it was seen in the past. To accomplish this efficiently, we describe a tree-based data structure that can generate such proofs with logarithmic size and space, improving over previous linear constructions. Where a classic hash chain might require an 800 MB trace to prove that a randomly chosen event is in a log with 80 million events, our prototype returns a 3 KB proof with the same semantics. We also present a flexible mechanism for the log server to present authenticated and tamper-evident search results for all events matching a predicate. This can allow large-scale log servers to selectively delete old events, in an agreed-upon fashion, while generating efficient proofs that no inappropriate events were deleted. We describe a prototype implementation and measure its performance on an 80 million event syslog trace at 1,750 events per second using a single CPU core. Performance improves to 10,500 events per second if cryptographic signatures are offloaded, corresponding to 1.1 TB of logging throughput per week.

1 Introduction

There are over 10,000 U.S. regulations that govern the storage and management of data [22, 58]. Many countries have legal, financial, medical, educational and privacy regulations that require businesses to retain a variety of records. Logging systems are therefore in wide use (albeit many without much in the way of security features).

Audit logs are useful for a variety of forensic purposes, such as tracing database tampering [59] or building a versioned filesystem with verifiable audit trails [52]. Tamper-evident logs have also been used to build Byzantine fault-tolerant systems [35] and protocols [15], as well as to detect misbehaving hosts in distributed systems [28].

Ensuring a log's integrity is a critical component in the security of a larger system. Malicious users, including insiders with high-level access and the ability to subvert the logging system, may want to perform unlogged activities or tamper with the recorded history. While tamper-resistance for such a system might be impossible, tamper-detection should be guaranteed in a strong fashion.

A variety of hash data structures have been proposed in the literature for storing data in a tamper-evident fashion, such as trees [34, 49], RSA accumulators [5, 11], skip lists [24], or general authenticated DAGs. These structures have been used to build certificate revocation lists [49], to build tamper-evident graph and geometric searching [25], and authenticated responses to XML queries [19]. All of these store static data, created by a *trusted author* whose signature is used as a root-of-trust for authenticating responses of a lookup queries.

While authenticated data structures have been adapted for dynamic data [2], they continue to assume a trusted author, and thus they have no need to detect inconsistencies across versions. For instance, in SUNDR [36], a trusted network filesystem is implemented on untrusted storage. Although version vectors [16] are used to detect when the server presents forking-inconsistent views to clients, only trusted clients sign updates for the filesystem.

Tamper-evident logs are fundamentally different: An *untrusted* logger is the sole author of the log and is responsible for both building and signing it. A log is a dynamic data structure, with the author signing a stream of commitments, a new commitment each time a new event is added to the log. Each commitment *snapshots* the entire log up to that point. If each signed commitment is the root of an authenticated data structure, well-known authenticated dictionary techniques [62, 42, 20] can detect tampering *within* each snapshot. However, without additional mechanisms to prevent it, an untrusted logger is free to have different snapshots make *inconsistent claims about the past*. To be secure, a tamper-evident log system must both detect tampering within each signed log *and* detect when different instances of the log make inconsistent claims.

Current solutions for detecting when an untrusted server is making inconsistent claims over time require linear space and time. For instance, to prevent undetected tampering, existing tamper evident logs [56, 17, 57] which rely upon a hash chain require auditors examine every intermediate event between snapshots. One proposal [43] for a tamper-evident log was based on a skip list. It has logarithmic lookup times, assuming the log

is known to be internally consistent. However, proving internal consistency requires scanning the full contents of the log. (See Section 3.4 for further analysis of this.)

In the same manner, CATS [63], a network-storage service with strong accountability properties, snapshots the internal state, and only probabilistically detects tampering by auditing a subset of objects for correctness between snapshots. Pavlou and Snodgrass [51] show how to integrate tamper-evidence into a relational database, and can prove the existence of tampering, if suspected. Auditing these systems for consistency is expensive, requiring each auditor visit each snapshot to confirm that any changes between snapshots are authorized.

If an untrusted logger knows that a just-added event or returned commitment will not be audited, then any tampering with the added event or the events fixed by that commitment will be undiscovered, and, by definition, the log is not tamper-evident. To prevent this, *a tamper-evident log requires frequent auditing*. To this end, we propose a tree-based history data structure, logarithmic for all auditing and lookup operations. Events may be added to the log, commitments generated, and audits may be performed independently of one another and at any time. No batching is used. Unlike past designs, we explicitly focus on how tampering will be discovered, through auditing, and we optimize the costs of these audits. Our *history tree* allows loggers to efficiently prove that the sequence of individual logs committed to, over time, make consistent claims about the past.

In Section 2 we present background material and propose semantics for tamper-evident logging. In Section 3 we present the history tree. In Section 4 we describe *Merkle aggregation*, a way to annotate events with attributes which can then be used to perform tamper-evident queries over the log and *safe deletion* of events, allowing unneeded events to be removed in-place, with no additional trusted party, while still being able to prove that no events were improperly purged. Section 5 describes a prototype implementation for tamper-evident logging of syslog data traces. Section 6 discusses approaches for scaling the logger's performance. Related work is presented in Section 7. Future work and conclusions appear in Section 8.

2 Security Model

In this paper, we make the usual cryptographic assumptions that an attacker cannot forge digital signatures or find collisions in cryptographic hash functions. Furthermore we are not concerned with protecting the secrecy of the logged events; this can be addressed with external techniques, most likely some form of encryption [50, 26, 54]. For simplicity, we assume a single monolithic log on a single host computer. Our goal is to detect tampering. It is impractical to prevent the destruction or alteration of digital records that are in the custody of a Byzantine logger. Replication strategies, outside the scope of this paper, can help ensure availability of the digital records [44].

Tamper-evidence requires auditing. If the log is never examined, then tampering cannot be detected. To this end, we divide a logging system into three logical entities—many *clients* which generate events for appending to a log or history, managed on a centralized but totally untrusted *logger*, which is ultimately audited by one or more trusted *auditors*. We assume clients and auditors have very limited storage capacity while loggers are assumed to have unlimited storage. By auditing the published commitments and demanding proofs, auditors can be convinced that the log's integrity has been maintained. At least one auditor is assumed to be incorruptible. In our system, we distinguish between clients and auditors, while a single host could, in fact, perform both roles.

We must trust clients to behave correctly while they are following the event insertion protocol, but we trust clients nowhere else. Of course, a malicious client could insert garbage, but we wish to ensure that an event, once correctly inserted, cannot be undetectably hidden or modified, even if the original client is subsequently colluding with the logger in an attempt to tamper with old data.

To ensure these semantics, an untrusted logger must regularly prove its correct behavior to auditors and clients. *Incremental proofs*, demanded of the logger, prove that current commitment and prior commitment make consistent claims about past events. *Membership proofs* ask the logger to return a particular event from the log along with a proof that the event is consistent with the current commitment. Membership proofs may be demanded by clients after adding events or by auditors verifying that older events remain correctly stored by the logger. These two styles of proofs are sufficient to yield tamper-evidence. As any vanilla lookup operation may be followed by a request for proof, the logger must behave faithfully or risk its misbehavior being discovered.

2.1 Semantics of a tamper evident history

We now formalize our desired semantics for secure histories. Each time an event X is sent to the logger, it assigns an index i and appends it to the log, generating a version-i commitment C_i that depends on all of the events to-date, $X_0 \ldots X_i$. The commitment C_i is bound to its version number i, signed, and published.

Although the stream of histories that a logger commits to $(C_0 \ldots C_i, C_{i+1}, C_{i+2} \ldots)$ are supposed to be mutually-consistent, each commitment fixes an *independent* history. Because histories are not known, a priori, to be consistent with one other, we will use primes ($'$) to distinguish between different histories and the events contained within them. In other words, the events in log C_i (i.e., those committed by commitment C_i) are $X_0 \ldots X_i$

and the events in log C'_j are $X'_0 \ldots X'_j$, and we will need to prove their correspondence.

2.1.1 Membership auditing

Membership auditing is performed both by clients, verifying that new events are correctly inserted, and by auditors, investigating that old events are still present and unaltered. The logger is given an event index i and a commitment C_j, $i \leq j$ and is required to return the ith element in the log, X_i, and a proof that C_j implies X_i is the ith event in the log.

2.1.2 Incremental auditing

While a verified membership proof shows that an event was logged correctly in *some* log, represented by its commitment C_j, additional work is necessary to verify that the sequence of logs committed by the logger is consistent over time. In *incremental auditing*, the logger is given two commitments C_j and C'_k, where $j \leq k$, and is required to prove that the two commitments make consistent claims about past events. A verified incremental proof demonstrates that $X_a = X'_a$ for all $a \in [0, j]$. Once verified, the auditor knows that C_j and C'_k commit to the same shared history, and the auditor can safely discard C_j.

A dishonest logger may attempt to tamper with its history by rolling back the log, creating a new fork on which it inserts new events, and abandoning the old fork. Such tampering will be caught if the logging system satisfies *historical consistency* (see Section 2.3) and by a logger's inability to generate an incremental proof between commitments on different (and inconsistent) forks when challenged.

2.2 Client insertion protocol

Once clients receive commitments from the logger after inserting an event, they must immediately redistribute them to auditors. This prevents the clients from subsequently colluding with the logger to roll back or modify their events. To this end, we need a mechanism, such as a gossip protocol, to distribute the signed commitments from clients to multiple auditors. It's unnecessary for every auditor to audit every commitment, so long as some auditor audits every commitment. (We further discuss tradeoffs with other auditing strategies in Section 3.1.)

In addition, in order to deal with the logger presenting different views of the log to different auditors and clients, auditors must obtain and reconcile commitments received from multiple clients or auditors, perhaps with the gossip protocol mentioned above. Alternatively the logger may publish its commitment in a public fashion so that all auditors receive the same commitment [27]. All that matters is that auditors have access to a diverse collection of commitments and demand incremental proofs to verify that the logger is presenting a consistent view.

2.3 Definition: tamper evident history

We now define a tamper-evident history system as a five-tuple of algorithms:

$H.\textsc{Add}(X) \rightarrow C_j$**.** Given an event X, appends it to the history, returning a new commitment.

$H.\textsc{Incr.Gen}(C_i, C_j) \rightarrow P$**.** Generates an incremental proof between C_i and C_j, where $i \leq j$.

$H.\textsc{Membership.Gen}(i, C_j) \rightarrow (P, X_i)$**.** Generates a membership proof for event i from commitment C_j, where $i \leq j$. Also returns the event, X_i.

$P.\textsc{Incr.Vf}(C'_i, C_j) \rightarrow \{\top, \bot\}$**.** Checks that P proves that C_j fixes every entry fixed by C'_i (where $i \leq j$). Outputs $\top$ if no divergence has been detected.

$P.\textsc{Membership.Vf}(i, C_j, X'_i) \rightarrow \{\top, \bot\}$**.** Checks that P proves that event X'_i is the i'th event in the log defined by C_j (where $i \leq j$). Outputs $\top$ if true.

The first three algorithms run on the logger and are used to append to the log H and to generate *proofs* P. Auditors or clients verify the proofs with algorithms $\{\textsc{Incr.Vf}, \textsc{Membership.Vf}\}$. Ideally, the proof P sent to the auditor is more concise than retransmitting the full history H. Only commitments need to be signed by the logger. Proofs do not require digital signatures; either they demonstrate consistency of the commitments and the contents of an event or they don't. With these five operations, we now define "tamper evidence" as a system satisfying:

Historical Consistency If we have a valid incremental proof between two commitments C_j and C_k, where $j \leq k$, ($P.\textsc{Incr.Vf}(C_j, C_k) \rightarrow \top$), and we have a valid membership proof P' for the event X'_i, where $i \leq j$, in the log fixed by C_j (i.e., $P'.\textsc{Membership.Vf}(i, C_j, X'_i) \rightarrow \top$) and a valid membership proof for X''_i in the log fixed by C_k (i.e., $P''.\textsc{Membership.Vf}(i, C_k, X''_i) \rightarrow \top$), then X'_i must equal X''_i. (In other words, if two commitments commit consistent histories, then they must both fix the same events for their shared past.)

2.4 Other threat models

Forward integrity Classic tamper-evident logging uses a different threat model, forward integrity [4]. The forward integrity threat model has two entities: clients who are fully trusted but have limited storage, and loggers who are assumed to be honest until suffering a Byzantine failure. In this threat model, the logger must be prevented from undetectably tampering with events logged prior to the Byzantine failure, but is allowed to undetectably tamper with events logged after the Byzantine failure.

Although we feel our threat model better characterizes the threats faced by tamper-evident logging, our history

tree and the semantics for tamper-evident logging are applicable to this alternative threat model with only minor changes. Under the semantics of forward-integrity, membership auditing just-added events is unnecessary because tamper-evidence only applies to events occurring before the Byzantine failure. Auditing a just-added event is unneeded if the Byzantine failure hasn't happened and irrelevant afterwards. Incremental auditing is still necessary. A client must incrementally audit received commitments to prevent a logger from tampering with events occurring before a Byzantine failure by rolling back the log and creating a new fork. Membership auditing is required to look up and examine old events in the log.

Itkis [31] has a similar threat model. His design exploited the fact that if a Byzantine logger attempts to roll back its history to before the Byzantine failure, the history must fork into two parallel histories. He proposed a procedure that tested two commitments to detect divergence without online interaction with the logger and proved an $O(n)$ lower bound on the commitment size. We achieve a tighter bound by virtue of the logger cooperating in the generation of these proofs.

Trusted hardware Rather than relying on auditing, an alternative model is to rely on the logger's hardware itself to be tamper-resistant [58, 1]. Naturally, the security of these systems rests on protecting the trusted hardware and the logging system against tampering by an attacker with complete physical access. Although our design could certainly use trusted hardware as an auditor, cryptographic schemes like ours rest on simpler assumptions, namely the logger can and must prove it is operating correctly.

3 History tree

We now present our new data structure for representing a tamper-evident history. We start with a Merkle tree [46], which has a long history of uses for authenticating static data. In a Merkle tree, data is stored at the leaves and the hash at the root is a tamper-evident summary of the contents. Merkle trees support logarithmic path lengths from the root to the leaves, permitting efficient random access. Although Merkle trees are a well-known tamper-evident data structure and our use is straightforward, the novelty in our design is in using a versioned computation of hashes over the Merkle tree to efficiently prove that different log snapshots, represented by Merkle trees, with *distinct* root hashes, make consistent claims about the past.

A filled history tree of depth d is a binary Merkle hash tree, storing 2^d events on the leaves. Interior nodes, $I_{i,r}$ are identified by their index i and layer r. Each leaf node $I_{i,0}$, at layer 0, stores event X_i. Interior node $I_{i,r}$ has left child $I_{i,r-1}$ and right child $I_{i+2^{r-1},r-1}$. (Figures 1 through 3 demonstrate this numbering scheme.) When a tree is not full, subtrees containing no events are represented as $\square$. This can be seen starting in Figure 1, a version-2 tree having three events. Figure 2 shows a version-6 tree, adding four additional events. Although the trees in our figures have a depth of 3 and can store up to 8 leaves, our design clearly extends to trees with greater depth and more leaves.

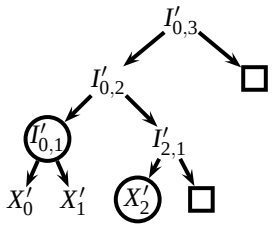

Figure 1: A version 2 history with commitment $C'_2 = I'_{0,3}$.

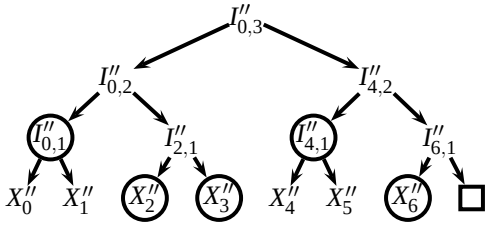

Figure 2: A version 6 history with commitment $C''_6 = I''_{0,3}$.

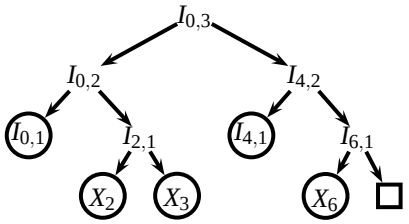

Figure 3: An incremental proof P between a version 2 and version 6 commitment. Hashes for the circled nodes are included in the proof. Other hashes can be derived from their children. Circled nodes in Figures 1 and 2 must be shown to be equal to the corresponding circled nodes here.

Each node in the history tree is *labeled* with a cryptographic hash which, like a Merkle tree, fixes the contents of the subtree rooted at that node. For a leaf node, the label is the hash of the event; for an interior node, the label is the hash of the concatenation of the labels of its children.

An interesting property of the history tree is the ability to efficiently reconstruct old versions or *views* of the tree. Consider the history tree given in Figure 2. The logger could reconstruct C''_2 analogous to the version-2 tree in Figure 1 by pretending that nodes $I''_{4,2}$ and X''_3 were $\square$ and then recomputing the hashes for the interior nodes and the root. If the reconstructed C''_2 matched a previously advertised commitment C'_2, then both trees must have the same contents and commit the same events.

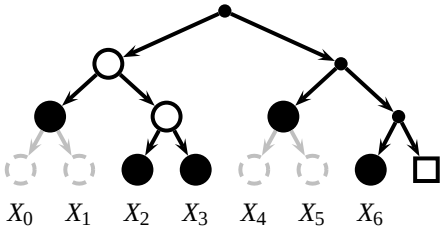

Figure 4: Graphical notation for a history tree analogous to the proof in Figure 3. Solid discs represent hashes included in the proof. Other nodes are not included. Dots and open circles represent values that can be recomputed from the values below them; dots may change as new events are added while open circles will not. Grey circle nodes are unnecessary for the proof.

This forms the intuition of how the logger generates an incremental proof P between two commitments, C_2' and C_6''. Initially, the auditor only possesses commitments C_2' and C_6''; it does not know the underlying Merkle trees that these commitments fix. The logger must show that both histories commit the same events, i.e., $X_0'' = X_0', X_1'' = X_1'$, and $X_2'' = X_2'$. To do this, the logger sends a *pruned tree* P to the auditor, shown in Figure 3. This pruned tree includes just enough of the full history tree to compute the commitments C_2 and C_6. Unnecessary subtrees are *elided* out and replaced with *stubs*. Events can be either included in the tree or replaced by a stub containing their hash. Because an incremental proof involves *three* history trees, the trees committed by C_2' and C_6'' with unknown contents and the pruned tree P, we distinguish them by using a different number of primes ($'$).

From P, shown in Figure 3, we reconstruct the corresponding root commitment for a version-6 tree, C_6. We recompute the hashes of interior nodes based on the hashes of their children until we compute the hash for node $I_{0,3}$, which will be the commitment C_6. If $C_6'' = C_6$ then the corresponding nodes, circled in Figures 2 and 3, in the pruned tree P and the implicit tree committed by C_6'' must match.

Similarly, from P, shown in Figure 3, we can reconstruct the version-2 commitment C_2 by pretending that the nodes X_3 and $I_{4,2}$ are $\square$ and, as before, recomputing the hashes for interior nodes up to the root. If $C_2' = C_2$, then the corresponding nodes, circled in Figures 1 and 3, in the pruned tree P and the implicit tree committed by C_2' must match, or $I_{0,1}' = I_{0,1}$ and $X_2' = X_2$.

If the events committed by C_2' and C_6'' are the same as the events committed by P, then they must be equal; we can then conclude that the tree committed by C_6'' is consistent with the tree committed by C_2'. By this we mean that the history trees committed by C_2' and C_6'' both commit the same events, or $X_0'' = X_0'$, $X_1'' = X_1'$, and $X_2'' = X_2'$, even though the events $X_0'' = X_0'$, $X_1'' = X_1'$, X_4'', and X_5'' are unknown to the auditor.

3.1 Is it safe to skip nodes during an audit?

In the pruned tree in Figure 3, we omit the events fixed by $I_{0,1}$, yet we still preserve the semantics of a tamper-evident log. Even though these earlier events may not be sent to the auditor, they are still fixed by the unchanged hashes above them in the tree. Any attempted tampering will be discovered in future incremental or membership audits of the skipped events. With the history tree, auditors only receive the portions of the history they need to audit the events they have chosen to audit. Skipping events makes it possible to conduct a variety of selective audits and offers more flexibility in designing auditing policies.

Existing tamper-evident log designs based on a classic hash-chain have the form $C_i = H(C_{i-1} \parallel X_i)$, $C_{-1} = \square$ and do not permit events to be skipped. With a hash chain, an incremental or membership proof between two commitments or between an event and a commitment must include *every* intermediate event in the log. In addition, because intermediate events cannot be skipped, each auditor, or client acting as an auditor, must eventually receive every event in the log. Hash chaining schemes, as such, are only feasible with low event volumes or in situations where every auditor is already receiving every event.

When membership proofs are used to investigate old events, the ability to skip nodes can lead to dramatic reductions in proof size. For example, in our prototype described in Section 5, in a log of 80 million events, our history tree can return a complete proof for any randomly chosen event in 3100 bytes. In a hash chain, where intermediate events cannot be skipped, an average of 40 million hashes would be sent.

Auditing strategies In many settings, it is possible that not every auditor will be interested in every logged event. Clients may not be interested in auditing events inserted or commitments received by other clients. One could easily imagine scenarios where a single logger is shared across many organizations, each only incentivized to audit the integrity of its own data. These organizations could run their own auditors, focusing their attention on commitments from their own clients, and only occasionally exchanging commitments with other organizations to ensure no forking has occurred. One can also imagine scenarios where independent accounting firms operate auditing systems that run against their corporate customers' log servers.

The log remains tamper-evident if clients gossip their received commitments from the logger to at least one honest auditor who uses it when demanding an incremental proof. By not requiring that every commitment be audited by every auditor, the total auditing overhead across all auditors can be proportional to the total number of events in the log—far cheaper than the number of events times the number of auditors as we might otherwise require.

$$A^v_{i,0} = \begin{cases} H(0 \,\|\, X_i) & \text{if } v \geq i \end{cases} \quad (1)$$

$$A^v_{i,r} = \begin{cases} H(1 \,\|\, A^v_{i,r-1} \,\|\, \square) & \text{if } v < i + 2^{r-1} \\ H(1 \,\|\, A^v_{i,r-1} \,\|\, A^v_{i+2^{r-1},r-1}) & \text{if } v \geq i + 2^{r-1} \end{cases} \quad (2)$$

$$C_n = A^n_{0,d} \quad (3)$$

$$A^v_{i,r} \equiv \mathrm{FH}_{i,r} \qquad \text{whenever } v \geq i + 2^r - 1 \quad (4)$$

Figure 5: Recurrence for computing hashes.

Skipping nodes offers other time-security tradeoffs. Auditors may conduct audits probabilistically, selecting only a subset of incoming commitments for auditing. If a logger were to regularly tamper with the log, its odds of remaining undetected would become vanishingly small.

3.2 Construction of the history tree

Now that we have an example of how to use a tree-based history, we will formally define its construction and semantics. A version-n history tree stores $n+1$ events, $X_0 \ldots X_n$. Hashes are computed over the history tree in a manner that permits the reconstruction of the hashes of interior nodes of older versions or *views*. We denote the hash on node $I_{i,r}$ by $A^v_{i,r}$ which is parametrized by the node's index, layer and view being computed. A version-v view on a version-n history tree reconstructs the hashes on interior nodes for a version-v history tree that only included events $X_0 \ldots X_v$. When $v = n$, the reconstructed root commitment is C_n. The hashes are computed with the recurrence defined in Figure 5.

A history tree can support arbitrary size logs by increasing the depth when the tree fills (i.e., $n = 2^d - 1$) and defining $d = \lceil \log_2(n+1) \rceil$. The new root, one level up, is created with the old tree as its left child and an empty right child where new events can be added. For simplicity in our illustrations and proofs, we assume a tree with fixed depth d.

Once a given subtree in the history tree is complete and has no more slots to add events, the hash for the root node of that subtree is *frozen* and will not change as future events are added to the log. The logger caches these frozen hashes (i.e., the hashes of frozen nodes) into $\mathrm{FH}_{i,r}$ to avoid the need to recompute them. By exploiting the frozen hash cache, the logger can recompute $A^v_{i,r}$ for any node with at most $O(d)$ operations. In a version-n tree, node $I_{i,r}$ is frozen when $n \geq i + 2^r - 1$. When inserting a new event into the log, $O(1)$ expected case and $O(d)$ worse case nodes will become frozen. (In Figure 1, node $I'_{0,1}$ is frozen. If event X_3 is added, nodes $I'_{2,1}$ and $I'_{0,2}$ will become frozen.)

Now that we have defined the history tree, we will describe the incremental proofs generated by the logger. Figure 4 abstractly illustrates a pruned tree equivalent to the proof given in Figure 3, representing an incremental proof from C_2 to C_6. Dots represent unfrozen nodes whose hashes are computed from their children. Open circles represent frozen nodes which are not included in the proof because their hashes can be recomputed from their children. Solid discs represent frozen nodes whose inclusion is necessary by being leaves or stubs. Grayed out nodes represent elided subtrees that are not included in the pruned tree. From this pruned tree and equations (1)-(4) (shown in Figure 5) we can compute $C_6 = A^6_{0,3}$ *and* a commitment from an earlier version-2 view, $A^2_{0,3}$.

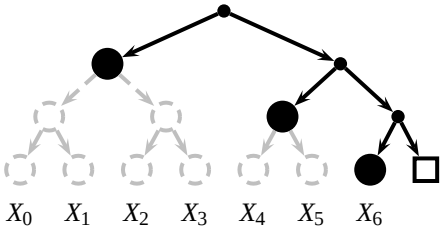

Figure 6: A proof skeleton for a version-6 history tree.

This pruned tree is incrementally built from a *proof skeleton*, seen in Figure 6—the minimum pruned tree of a version-6 tree consisting only of frozen nodes. The proof skeleton for a version-n tree consists of frozen hashes for the left siblings for the path from X_n to the root. From the included hashes and using equations (1)-(4), this proof skeleton suffices to compute $C_6 = A^6_{0,3}$.

From Figure 6 the logger incrementally builds Figure 4 by splitting frozen interior nodes. A node is split by including its children's hashes in the pruned tree instead of itself. By recursively splitting nodes on the path to a leaf, the logger can *include* that leaf in the pruned tree. In this example, we split nodes $I_{0,2}$ and $I_{2,1}$. For each commitment C_i that is to be reconstructable in an incremental proof the pruned tree P must include a path to the event X_i. The same algorithm is used to generate the membership proof for an event X_i.

Given these constraints, we can now define the five history operations in terms of the equations in Figure 5.

$H.\textsc{Add}(X) \to C_n$**.** Event is assigned the next free slot, n. C_n is computed by equations (1)-(4).

$H.\textsc{Incr.Gen}(C_i, C_j) \to P$**.** The pruned tree P is a version-j proof skeleton including a path to X_i.

$H.\textsc{Membership.Gen}(i, C_j) \to (P, X_i)$**.** The pruned tree P is a version-j proof skeleton including a path to X_i.

$P.\textsc{Incr.Vf}(C''_i, C'_j) \to \{\top, \bot\}$**.** From P apply equations (1)-(4) to compute $A^i_{0,d}$ and $A^j_{0,d}$. This can only be done if P includes a path to the leaf X_i. Return $\top$ if $C''_i = A^i_{0,d}$ and $C'_j = A^j_{0,d}$.

P.MEMBERSHIP.VF$(i, C'_j, X'_i) \rightarrow \{\top, \bot\}$. From P apply equations (1)-(4) to compute $A^j_{0,d}$. Also extract X_i from the pruned tree P, which can only be done if P includes a path to event X_i. Return $\top$ if $C'_j = A^j_{0,d}$ and $X_i = X'_i$.

Although incremental and membership proofs have different semantics, they both follow an identical tree structure and can be built and audited by a common implementation. In addition, a single pruned tree P can embed paths to several leaves to satisfy multiple auditing requests.

What is the size of a pruned tree used as a proof? The pruned tree necessary for satisfying a self-contained incremental proof between C_i and C_j or a membership proof for i in C_j requires that the pruned tree include a path to nodes X_i and X_j. This resulting pruned tree contains at most $2d$ frozen nodes, logarithmic in the size of the log.

In a real implementation, the log may have moved on to a later version, k. If the auditor requested an incremental proof between C_i and C_j, the logger would return the latest commitment C_k, and a pruned tree of at most $3d$ nodes, based around a version-k tree including paths to X_i and X_j. More typically, we expect auditors will request an incremental proof between a commitment C_i and the latest commitment. The logger can reply with the latest commitment C_k and pruned tree of at most $2d$ nodes that included a path to X_i.

The frozen hash cache In our description of the history tree, we described the *full representation* when we stated that the logger stores frozen hashes for all frozen interior nodes in the history tree. This cache is redundant whenever a node's hash can be recomputed from its children. We expect that logger implementations, which build pruned trees for audits and queries, will maintain and use the cache to improve efficiency.

When generating membership proofs, incremental proofs, and query lookup results, there is no need for the resulting pruned tree to include redundant hashes on interior nodes when they can be recomputed from their children. We assume that pruned trees used as proofs will use this *minimum representation*, containing frozen hashes only for stubs, to reduce communication costs.

Can overheads be reduced by exploiting redundancy between proofs? If an auditor is in regular communication with the logger, demanding incremental proofs between the previously seen commitment and the latest commitment, there is redundancy between the pruned subtrees on successive queries.

If an auditor previously requested an incremental proof between C_i and C_j and later requests an incremental proof P between C_j and C_n, the two proofs will share hashes on the path to leaf X_j. The logger may send a *partial proof* that omits these common hashes, and only contains the expected $O(\log_2(n-j))$ frozen hashes that are not shared between the paths to X_j and X_n. This devolves to $O(1)$ if a proof is requested after every insertion. The auditor need only cache d frozen hashes to make this work.

Tree history time-stamping service Our history tree can be adapted to implement a round-based time-stamping service. After every round, the logger publishes the last commitment in public medium such as a newspaper. Let C_i be the commitment from the prior round and C_k be the commitment of the round a client requests that its document X_j be timestamped. A client can request a pruned tree including a path to leaves X_i, X_j, X_k. The pruned tree can be verified against the published commitments to prove that X_j was submitted in the round and its order within that round, without the cooperation of the logger.

If a separate history tree is built for each round, our history tree is equivalent to the threaded authentication tree proposed by Buldas et al. [10] for time-stamping systems.

3.3 Storing the log on secondary storage

Our history tree offers a curious property: it can be easily mapped onto write-once append-only storage. Once nodes become frozen, they become immutable, and are thus safe to output. This ordering is predetermined, starting with (X_0), $(X_1, I_{0,1})$, (X_2), $(X_3, I_{2,1}, I_{0,2})$, $(X_4)\ldots$. Parentheses denote the nodes written by each ADD transaction. If nodes within each group are further ordered by their layer in the tree, this order is simply a post-order traversal of the binary tree. Data written in this linear fashion will minimize disk seek overhead, improving the disk's write performance. Given this layout, and assuming all events are the same size on disk, converting from an $(index, layer)$ to the byte index used to store that node takes $O(\log n)$ arithmetic operations, permitting efficient direct access.

In order to handle variable-length events, event data can be stored in a separate write-once append-only *value store*, while the leaves of the history tree contain offsets into the value store where the event contents may be found. Decoupling the history tree from the value store also allows many choices for how events are stored, such as databases, compressed files, or standard flat formats.

3.4 Comparing to other systems

In this section, we evaluate the time and space tradeoffs between our history tree and earlier hash chain and skip list structures. In all three designs, membership proofs have the same structure and size as incremental proofs, and proofs are generated in time proportional to their size.

Maniatis and Baker [43] present a tamper-evident log using a deterministic variant of a skip list [53]. The skip list history is like a hash-chain incorporating extra skip links that hop over many nodes, allowing for logarithmic lookups.

	Hash chain	Skip list	History tree
ADD Time	$O(1)$	$O(1)$	$O(\log_2 n)$
INCR.GEN proof size to C_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
MEMBERSHIP.GEN proof size for X_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
Cache size	-	$O(\log_2 n)$	$O(\log_2 n)$
INCR.GEN partial proof size	-	$O(n-j)$	$O(\log_2(n-j))$
MEMBERSHIP.GEN partial proof size	-	$O(\log_2(n-i))$	$O(\log_2(n-i))$

Table 1: We characterize the time to add an event to the log and the size of full and partial proofs generated in terms of n, the number of events in the log. For partial proofs audits, j denotes the number of events in the log at the time of the last audit and i denotes the index of the event being membership-audited.

In Table 1 we compare the three designs. All three designs have $O(1)$ storage per event and $O(1)$ commitment size. For skip list histories and tree histories, which support partial proofs (described in Section 3.2), we present the cache size and the expected proof sizes in terms of the number of events in the log, n, and the index, j, of the prior contact with the logger or the index i of the event being looked up. Our tree-based history strictly dominates both hash chains and skip lists in proof generation time and proof sizes, particularly when individual clients and auditors only audit a subset of the commitments or when partial proofs are used.

Canonical representation A hash chain history and our history tree have a canonical representation of both the history and of proofs within the history. In particular, from a given commitment C_n, there exists one unique path to each event X_i. When there are multiple paths auditing is more complex because the alternative paths must be checked for consistency with one another, both within a single history, and between the stream of histories $C_i, C_{i+1}, \ldots$ committed by the logger. Extra paths may improve the efficiency of looking up past events, such as in a skip list, or offer more functionality [17], but cannot be trusted by auditors and must be checked.

Maniatis and Baker [43] claim to support logarithmic-sized proofs, however they suffer from this multi-path problem. To verify internal consistency, an auditor with no prior contact with the logger must receive every event in the log in every incremental or membership proof.

Efficiency improves for auditors in regular contact with the logger that use partial proofs and cache $O(\log_2 n)$ state between incremental audits. If an auditor has previously verified the logger's internal consistency up to C_j, the auditor will be able to verify the logger's internal consistency up to a future commitment C_n with the receipt of events $X_{j+1} \ldots X_n$ Once an auditor knows that the skip list is internally consistent the links that allow for logarithmic lookups can be trusted and subsequent membership proofs on old events will run in $O(\log_2 n)$ time. Skip list histories were designed to function in this mode, with each auditor eventually receiving every event in the log.

Auditing is required Hash chains and skip lists only offer a complexity advantage over the history tree when adding new events, but this advantage is fleeting. If the logger knows that a given commitment will never be audited, it is free to tamper with the events fixed by that commitment, and the log is no longer provably tamper evident. Every commitment returned by the logger must have a non-zero chance of being audited and any evaluation of tamper-evident logging must include the costs of this unavoidable auditing. With multiple auditors, auditing overhead is further multiplied. After inserting an event, hash chains and skip lists suffer an $O(n-j)$ disadvantage the moment they do incremental audits between the returned commitment and prior commitments. They cannot reduce this overhead by, for example, only auditing a random subset of commitments.

Even if the threat model is weakened from our always-untrusted logger to the forward-integrity threat model (See Section 2.4), hash chains and skip lists are less efficient than the history tree. Clients can forgo auditing just-added events, but are still required to do incremental audits to prior commitments, which are expensive with hash chains or skip lists.

4 Merkle aggregation

Our history tree permits $O(\log_2 n)$ access to arbitrary events, given their index. In this section, we extend our history tree to support efficient, tamper-evident content searches through a feature we call *Merkle aggregation*, which encodes auxiliary information into the history tree. Merkle aggregation permits the logger to perform authorized purges of the log while detecting unauthorized deletions, a feature we call *safe deletion*.

As an example, imagine that a client flags certain events in the log as "important" when it stores them. In the history tree, the logger propagates these flags to interior nodes, setting the flag whenever either child is flagged. To ensure that the tagged history is tamper-evident, this flag can be incorporated into the hash label of a node and checked during auditing. As clients are assumed to be trusted when inserting into the log, we assume clients will properly annotate their events. Membership auditing will detect if the logger incorrectly stored a leaf with the wrong flag or improperly propagated the flag. Incremental audits would detect tampering if any frozen

node had its flag altered. Now, when an auditor requests a list of only flagged events, the logger can generate that list along with a proof that the list is complete. If there are relatively few "important" events, the query results can skip over large chunks of the history.

To generate a proof that the list of flagged events is complete, the logger traverses the full history tree H, pruning any subtrees without the flag set, and returns a pruned tree P containing only the visited nodes. The auditor can ensure that no flagged nodes were omitted in P by performing its own recursive traversal on P and verifying that every stub is unflagged.

Figure 7 shows the pruned tree for a query against a version-5 history with events X_2 and X_5 flagged. Interior nodes in the path from X_2 and X_5 to the root will also be flagged. For subtrees containing no matching events, such as the parent of X_0 and X_1, we only need to retain the root of the subtree to vouch that its children are unflagged.

4.1 General attributes

Boolean flags are only one way we may flag log events for later queries. Rather than enumerate every possible variation, we abstract an aggregation strategy over attributes into a 3-tuple, $(\tau, \oplus, \Gamma)$. τ represents the type of attribute or attributes that an event has. $\oplus$ is a deterministic function used to compute the attributes on an interior node in the history tree by *aggregating* the attributes of the node's children. Γ is a deterministic function that maps an event to its attributes. In our example of client-flagged events, the aggregation strategy is $(\tau := \text{BOOL}, \oplus := \vee, \Gamma(x) := x.isFlagged)$.

For example, in a banking application, an attribute could be the dollar value of a transaction, aggregated with the MAX function, permitting queries to find all transactions over a particular dollar value and detect if the logger tampers with the results. This corresponds to $(\tau := \text{INT}, \oplus := \text{MAX}, \Gamma(x) := x.value)$. Or, consider events having internal timestamps, generated by the client, arriving at the logger out of order. If we attribute each node in the tree with the earliest and latest timestamp found among its children, we can now query the logger for all nodes within a given time range, regardless of the order of event arrival.

There are at least three different ways to implement keyword searching across logs using Merkle aggregation. If the number of keywords is fixed in advance, then the attribute τ for events can be a bit-vector or sparse bit-vector combined with $\oplus := \vee$. If the number of keywords is unknown, but likely to be small, τ can be a sorted list of keywords, with $\oplus := \cup$ (set union). If the number of keywords is unknown and potentially unbounded, then a Bloom filter [8] may be used to represent them, with τ being a bit-vector and $\oplus := \vee$. Of course, the Bloom filter would then have the potential of returning false positives to a query, but there would be no false negatives.

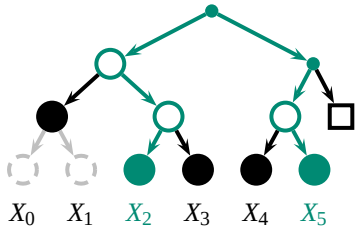

Figure 7: Demonstration of Merkle aggregation with some events flagged as important (highlighted). Frozen nodes that would be included in a query are represented as solid discs.

Merkle aggregation is extremely flexible because Γ can be *any* deterministic computable function. However, once a log has been created, $(\tau, \oplus, \Gamma)$ are fixed for that log, and the set of queries that can be made is restricted based on the aggregation strategy chosen. In Section 5 we describe how we were able to apply these concepts to the metadata used in Syslog logs.

4.2 Formal description

To make attributes tamper-evident in history trees, we modify the computation of hashes over the tree to include them. Each node now has a hash label denoted by $A^v_{i,r}.H$ and an annotation denoted by $A^v_{i,r}.A$ for storing attributes. Together these form the node data that is attached to each node in the history tree. Note that the hash label of node, $A^v_{i,r}.H$, does *not* fix its own attributes, $A^v_{i,r}.A$. Instead, we define a *subtree authenticator* $A^v_{i,r}.* = H(A^v_{i,r}.H \parallel A^v_{i,r}.A)$ that fixes the attributes and hash of a node, and recursively fixes every hash and attribute in its subtree. Frozen hashes $\text{FH}_{i,r}.A$ and $\text{FH}_{i,r}.H$ and $\text{FH}_{i,r}.*$ are defined analogously to the non-Merkle-aggregation case.

We could have defined this recursion in several different ways. This representation allows us to elide unwanted subtrees with a small stub, containing one hash and one set of attributes, while exposing the attributes in a way that makes it possible to locally detect if the attributes were improperly aggregated.

Our new mechanism for computing hash and aggregates for a node is given in equations (5)-(10) in Figure 8. There is a strong correspondence between this recurrence and the previous one in Figure 5. Equations (6) and (7) extract the hash and attributes of an event, analogous to equation (1). Equation (9) handles aggregation of attributes between a node and its children. Equation (8) computes the hash of a node in terms of the subtree authenticators of its children.

INCR.GEN and MEMBERSHIP.GEN operate the same as with an ordinary history tree, except that wherever a frozen hash was included in the proof ($\text{FH}_{i,r}$), we now include both the hash of the node, $\text{FH}_{i,r}.H$, and its attributes $\text{FH}_{i,r}.A$. Both are required for recomputing $A^v_{i,r}.A$ and $A^v_{i,r}.H$ for the parent node. ADD, INCR.VF,

$$A^v_{i,r}.* = H(A^v_{i,r}.H \,\|\, A^v_{i,r}.A) \tag{5}$$

$$A^v_{i,0}.H = \begin{cases} H(0 \,\|\, X_i) & \text{if } v \geq i \end{cases} \tag{6}$$

$$A^v_{i,0}.A = \begin{cases} \Gamma(X_i) & \text{if } v \geq i \end{cases} \tag{7}$$

$$A^v_{i,r}.H = \begin{cases} H(1 \,\|\, A^v_{i,r-1}.* \,\|\, \square) & \text{if } v < i + 2^{r-1} \\ H(1 \,\|\, A^v_{i,r-1}.* \,\|\, A^v_{i+2^{r-1},r-1}.*) & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{8}$$

$$A^v_{i,r}.A = \begin{cases} A^v_{i,r-1}.A & \text{if } v < i + 2^{r-1} \\ A^v_{i,r-1}.A \oplus A^v_{i+2^{r-1},r-1}.A & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{9}$$

$$C_n = A^n_{0,d}.* \tag{10}$$

Figure 8: Hash computations for Merkle aggregation

and MEMBERSHIP.VF are the same as before except for using the equations (5)-(10) for computing hashes and propagating attributes. Merkle aggregation inflates the storage and proof sizes by a factor of $(A+B)/A$ where A is the size of a hash and B is the size of the attributes.

4.2.1 Queries over attributes

In Merkle aggregation queries, we permit query results to contain false positives, i.e., events that do not match the query Q. Extra false positive events in the result only impact performance, not correctness, as they may be filtered by the auditor. We forbid false negatives; every event matching Q will be included in the result.

Unfortunately, Merkle aggregation queries can only match attributes, not events. Consequently, we must conservatively transform a query Q over events into a predicate Q^Γ over attributes and require that it be *stable*, with the following properties: If Q matches an event then Q^Γ matches the attributes of that event (i.e., $\forall_x\ Q(x) \Rightarrow Q^\Gamma(\Gamma(x))$). Furthermore, if Q^Γ is true for either child of a node, it must be true for the node itself (i.e., $\forall_{x,y}\ Q^\Gamma(x) \vee Q^\Gamma(y) \Rightarrow Q^\Gamma(x \oplus y)$ and $\forall_x\ Q^\Gamma(x) \vee Q^\Gamma(\square) \Rightarrow Q^\Gamma(x \oplus \square)$).

Stable predicates can falsely match nodes or events for two reasons: events' attributes may match Q^Γ without the events matching Q, or nodes may occur where $(Q^\Gamma(x) \vee Q^\Gamma(y))$ is false, but $Q^\Gamma(x \oplus y)$ is true. We call a predicate Q *exact* if there can be no false matches. This occurs when $Q(x) \Leftrightarrow Q^\Gamma(\Gamma(x))$ and $Q^\Gamma(x) \vee Q^\Gamma(y) \Leftrightarrow Q^\Gamma(x \oplus y)$. Exact queries are more efficient because a query result does not include falsely matching events and the corresponding pruned tree proving the correctness of the query result does not require extra nodes.

Given these properties, we can now define the additional operations for performing authenticated queries on the log for events matching a predicate Q^Γ.

$H.\text{QUERY}(C_j, Q^\Gamma) \to P$ Given a predicate Q^Γ over attributes τ, returns a pruned tree where every elided subtrees does not match Q^Γ.

$P.\text{QUERY.VF}(C'_j, Q^\Gamma) \to \{\top, \bot\}$ Checks the pruned tree P and returns $\top$ if every stub in P does not match Q^Γ and the reconstructed commitment C_j is the same as C'_j.

Building a pruned tree containing all events matching a predicate Q^Γ is similar to building the pruned trees for membership or incremental auditing. The logger starts with a proof skeleton then recursively traverses it, splitting interior nodes when $Q^\Gamma(FH_{i,r}.A)$ is true. Because the predicate Q^Γ is stable, no event in any elided subtree can match the predicate. If there are t events matching the predicate Q^Γ, the pruned tree is of size at most $O((1+t)\log_2 n)$ (i.e., t leaves with $\log_2 n$ interior tree nodes on the paths to the root).

To verify that P includes all events matching Q^Γ, the auditor does a recursive traversal over P. If the auditor finds an interior stub where $Q^\Gamma(FH_{i,r}.A)$ is true, the verification fails because the auditor found a node that was supposed to have been split. (Unfrozen nodes will always be split as they compose the proof skeleton and only occur on the path from X_j to the root.) The auditor must also verify that pruned tree P commits the same events as the commitment C'_j by reconstructing the root commitment C_j using the equations (5)-(10) and checking that $C_j = C'_j$.

As with an ordinary history tree, a Merkle aggregating tree requires auditing for tamper-detection. If an event is never audited, then there is no guarantee that its attributes have been properly included. Also, a dishonest logger or client could deliberately insert false log entries whose attributes are aggregated up the tree to the root, causing garbage results to be included in queries. Even so, if Q is stable, a malicious logger cannot hide matching events from query results without detection.

4.3 Applications

Safe deletion Merkle aggregation can be used for expiring old and obsolete events that do not satisfy some predicate and prove that no other events were deleted inappropriately. While Merkle aggregation queries prove that no matching event is excluded from a query result, safe deletion requires the contrapositive: proving to an auditor that each purged event was legitimately purged because it did not match the predicate.

Let $Q(x)$ be a stable query that is true for all events that the logger must keep. Let $Q^\Gamma(x)$ be the corresponding predicate over attributes. The logger stores a pruned tree that includes all nodes and leaf events where $Q^\Gamma(x)$ is true. The remaining nodes may be elided and replaced with stubs. When a logger cannot generate a path to a previously deleted event X_i, it instead supplies a pruned tree that includes a path to an ancestor node A of X_i where $Q^\Gamma(A)$ is false. Because Q is stable, if $Q^\Gamma(A)$ is false, then $Q^\Gamma(\Gamma(X_i))$ and $Q(X_i)$ must also be false.

Safe deletion and auditing policies must take into account that if a subtree containing events $X_i \ldots X_j$ is purged, the logger is unable to generate incremental or membership proofs involving commitments $C_i \ldots C_j$. The auditing policy must require that any audits using those commitments be performed before the corresponding events are deleted, which may be as simple as requiring that clients periodically request an incremental proof to a later or long-lived commitment.

Safe deletion will not save space when using the append-only storage described in Section 3.3. However, if data-destruction policies require destroying a subset of events in the log, safe deletion may be used to prove that no unauthorized log events were destroyed.

"Private" search Merkle aggregation enables a weak variant of private information retrieval [14], permitting clients to have privacy for the specific contents of their events. To aggregate the attributes of an event, the logger only needs the attributes of an event, $\Gamma(X_i)$, not the event itself. To verify that aggregation is done correctly also only requires the attributes of an event. If clients encrypt their events and digitally sign their public attributes, auditors may verify that aggregation is done correctly while clients preserve their event privacy from the logger and other clients and auditors.

Bloom filters, in addition to providing a compact and approximate way to represent the presence or absence of a large number of keywords, can also enable private indexing (see, e.g., Goh [23]). The logger has no idea what the individual keywords are within the Bloom filter; many keywords could map to the same bit. This allows for private keywords that are still protected by the integrity mechanisms of the tree.

5 Syslog prototype implementation

Syslog is the standard Unix-based logging system [38], storing events with many attributes. To demonstrate the effectiveness of our history tree, we built an implementation capable of storing and searching syslog events. Using events from syslog traces, captured from our departmental servers, we evaluated the storage and performance costs of tamper-evident logging and secure deletion.

Each syslog event includes a timestamp, the host generating the event, one of 24 *facilities* or subsystem that generated the event, one of 8 logging *levels*, and the *message*. Most events also include a *tag* indicating the program generating the event. Solutions for authentication, management, and reliable delivery of syslog events over the network have already been proposed [48] and are in the process of being standardized [32], but none of this work addresses the logging semantics that we wish to provide.

Our prototype implementation was written in a hybrid of Python 2.5.2 and C++ and was benchmarked on an Intel Core 2 Duo 2.4GHz CPU with 4GB of RAM in 64-bit mode under Linux. Our present implementation is single-threaded, so the second CPU core is underutilized. Our implementation uses SHA-1 hashes and 1024-bit DSA signatures, borrowed from the OpenSSL library.

In our implementation, we use the array-based post-order traversal representation discussed in Section 3.3. The value store and history tree are stored in separate write-once append-only files and mapped into memory. Nodes in the history tree use a fixed number of bytes, permitting direct access. Generating membership and incremental proofs requires RAM proportional to the size of the proof, which is logarithmic in the number of events in the log. Merkle aggregation query result sizes are presently limited to those which can fit in RAM, approximately 4 million events.

The storage overheads of our tamper-evident history tree are modest. Our prototype stores five attributes for each event. Tags and host names are encoded as 2-of-32 bit Bloom filters. Facilities and hosts are encoded as bit-vectors. To permit range queries to find every event in a particular range of time, an interval is used to encode the message timestamp. All together, there are twenty bytes of attributes and twenty bytes for a SHA-1 hash for each node in the history tree. Leaves have an additional twelve bytes to store the offset and length of the event contents in the value store.

We ran a number of simulations of our prototype to determine the processing time and space overheads of the history tree. To this end, we collected a trace of four million events from thirteen of our departmental server hosts over 106 hours. We observed 9 facilities, 6 levels, and 52 distinct tags. 88.1% of the events are from the mail server and 11.5% are from 98,743 failed ssh connection attempts. Only .393% of the log lines are from other sources. In testing our history tree, we replay this trace 20 times to insert 80 million events. Our syslog trace, after the replay, occupies 14.0 GB, while the history tree adds an additional 13.6 GB.

5.1 Performance of the logger

The logger is the only centralized host in our design and may be a bottleneck. The performance of a real world logger will depend on the auditing policy and relative frequency between inserting events and requesting audits. Rather than summarize the performance of the logger for one particular auditing policy, we benchmark the costs of the various tasks performed by the logger.

Our captured syslog traces averaged only ten events per second. Our prototype can insert events at a rate of 1,750 events per second, including DSA signature generation. Inserting an event requires four steps, shown in Table 2, with the final step, signing the resulting commitment, responsible for most of the processing time. Throughput

Step	Task	% of CPU	Rate (events/sec)
A	Parse syslog message	2.4%	81,000
B	Insert event into log	2.6%	66,000
C	Generate commitment	11.8%	15,000
D	Sign commitment	83.3%	2,100
	Membership proofs (with locality)	-	8,600
	Membership proofs (no locality)	-	32

Table 2: Performance of the logger in each of the four steps required to insert an event and sign the resulting commitment and in generating membership proofs. Rates are given assuming nothing other than the specified step is being performed.

would increase to 10,500 events per second if the DSA signatures were computed elsewhere (e.g., leveraging multiple CPU cores). (Section 6 discusses scalability in more detail.) This corresponds to 1.9MB/sec of uncompressed syslog data (1.1 TB per week).

We also measured the rate at which our prototype can generate membership and incremental proofs. The size of an incremental proof between two commitments depends upon the distance between the two commitments. As the distance varies from around two to two million events, the size of a self-contained proof varies from 1200 bytes to 2500 bytes. The speed for generating these proofs varies from 10,500 proofs/sec to 18,000 proofs/sec, with shorter distances having smaller proof sizes and faster performance than longer distances. For both incremental and membership proofs, compressing by gzip [18] halves the size of the proofs, but also halves the rate at which proofs can be generated.

After inserting 80 million events into the history tree, the history tree and value store require 27 GB, several times larger than our test machine's RAM capacity. Table 2 presents our results for two membership auditing scenarios. In our first scenario we requested membership proofs for random events chosen among the most recent 5 million events inserted. Our prototype generated 8,600 self-contained membership proofs per second, averaging 2,400 bytes each. In this high-locality scenario, the most recent 5 million events were already sitting in RAM. Our second scenario examined the situation when audit requests had low locality by requesting membership proofs for random events anywhere in the log. The logger's performance was limited to our disk's seek latency. Proof size averaged 3,100 bytes and performance degraded to 32 membership proofs per second. (We discuss how this might be overcome in Section 6.2.)

To test the scalability of the history tree, we benchmarked insert performance and auditing performance on our original 4 million event syslog event trace, without replication, and the 80 million event trace after 20x replication. Event insertion and incremental auditing are roughly 10% slower on the larger log.

5.2 Performance of auditors and clients

The history tree places few demands upon auditors or clients. Auditors and clients must verify the logger's commitment signatures and must verify the correctness of pruned tree replies to auditing requests. Our machine can verify 1,900 DSA-1024 signatures per second. Our current tree parser is written in Python and is rather slow. It can only parse 480 pruned trees per second. Once the pruned tree has been parsed, our machine can verify 9,000 incremental or membership proofs per second. Presently, one auditor cannot verify proofs as fast as the logger can generate them, but auditors can clearly operate independently of one another, in parallel, allowing for exceptional scaling, if desired.

5.3 Merkle aggregation results

In this subsection, we describe the benefits of Merkle aggregation in generating query results and in safe deletion. In our experiments, due to limitations of our implementation in generating large pruned trees, our Merkle aggregation experiments used the smaller four million event log.

We used 86 different predicates to investigate the benefits of safe deletion and the overheads of Merkle aggregation queries. We used 52 predicates, each matching one tag, 13 predicates, each matching one host, 9 predicates, each matching one facility, 6 predicates, one matching each level, and 6 predicates, each matching the k highest logging levels.

The predicates matching tags and hosts use Bloom filters, are *inexact*, and may have false positives. This causes 34 of the 65 Bloom filter query results to include more nodes than our "worst case" expectation for exact predicates. By using larger Bloom filters, we reduce the chances of spurious matches. When a 4-of-64 Bloom filter is used for tags and hostnames, pruned trees resulting from search queries average 15% fewer nodes, at the cost of an extra 64 bits of attributes for each node in the history tree. In a real implementation, the exact parameters of the Bloom filter would best be tuned to match a sample of the events being logged.

Merkle aggregation and safe deletion Safe deletion allows the purging of unwanted events from the log. Auditors define a stable predicate over the attributes of events indicating which events must be kept, and the logger keeps a pruned tree of only those matching events. In our first test, we simulated the deletion of all events except those from a particular host. The pruned tree was generated in 14 seconds, containing 1.92% of the events in the full log and serialized to 2.29% of the size of the full tree. Although 98.08% of the events were purged, the logger was only able to purge 95.1% of the nodes in the

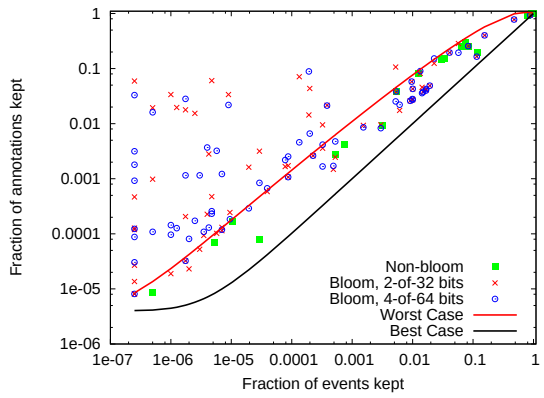

Figure 9: Safe deletion overhead. For a variety of queries, we plot the fraction of hashes and attributes kept after deletion versus the fraction of events kept.

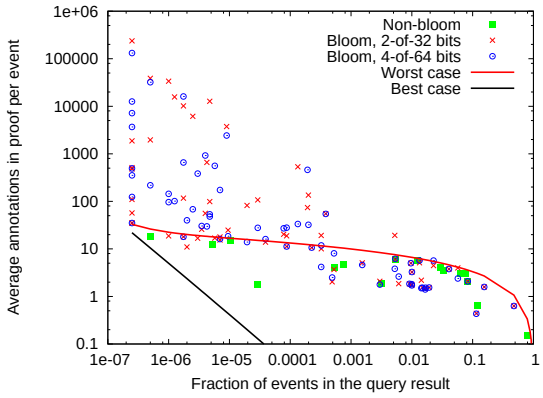

Figure 10: Query overhead per event. We plot the ratio between the number of hashes and matching events in the result of each query versus the fraction of events matching the query.

history tree because the logger must keep the hash label and attributes for the root nodes of elided subtrees.

When measuring the size of a pruned history tree generated by safe deletion, we assume the logger caches hashes and attributes for all interior nodes in order to be able to quickly generate proofs. For each predicate, we measure the *kept ratio*, the number of interior node or stubs in a pruned tree of all nodes matching the predicate divided by the number of interior nodes in the full history tree. In Figure 9 for each predicate we plot the kept ratio versus the fraction of events matching the predicate. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. Our Bloom-filter queries do worse than the "worst-case" bound because Bloom filter matches are inexact and will thus trigger false positive matches on interior nodes, forcing them to be kept in the resulting pruned tree. Although many Bloom filters did far worse than the "worst-case," among the Bloom filters that matched fewer than 1% of the events in the log, the logger is still able to purge over 90% of the nodes in the history tree and often did much better than that.

Merkle aggregation and authenticated query results In our second test, we examine the overheads for Merkle aggregation query lookup results. When the logger generates the results to a query, the resulting pruned tree will contain both matching events and history tree overhead, in the form of hashes and attributes for any stubs. For each predicate, we measure the *query overhead ratio*—the number of stubs and interior nodes in a pruned tree divided by the number of events in the pruned tree. In Figure 10 we plot the query overhead ratio versus the fraction of events matching the query for each of our 86 predicates. This plot shows, for each event matching a predicate, proportionally how much extra overhead is incurred, per event, for authentication information. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. With exact predicates, the overhead of authenticated query results is very modest, and again, inexact Bloom filter queries will sometimes do worse than the "worst case."

6 Scaling a tamper-evident log

In this section, we discuss techniques to improve the insert throughput of the history tree by using concurrency, and to improve the auditing throughput with replication. We also discuss a technique to amortize the overhead of a digital signature over several events.

6.1 Faster inserts via concurrency

Our tamper-evident log offers many opportunities to leverage concurrency to increase throughput. Perhaps the simplest approach is to offload signature generation. From Table 2, signatures account for over 80% of the runtime cost of an insert. Signatures are not included in any other hashes and there are no interdependencies between signature computations. Furthermore, signing a commitment does not require knowing anything other than the root commitment of the history tree. Consequently, it's easy to offload signature computations onto additional CPU cores, additional hosts, or hardware crypto accelerators to improve throughput.

It is possible for a logger to also generate commitments concurrently. If we examine Table 2, parsing and inserting events in the log is about two times faster than generating commitments. Like signatures, commitments have no interdependencies on one other; they depend only on the history tree contents. As soon as event X_j is inserted into the tree and $O(1)$ frozen hashes are computed and stored,

a new event may be immediately logged. Computing the commitment C_j only requires read-only access to the history tree, allowing it to be computed concurrently by another CPU core without interfering with subsequent events. By using shared memory and taking advantage of the append-only write-once semantics of the history tree, we would expect concurrency overhead to be low.

We have experimentally verified the maximum rate at which our prototype implementation, described in Section 5, can insert syslog events into the log at 38,000 events per second using only one CPU core on commodity hardware. This is the maximum throughput our hardware could potentially support. In this mode we assume that digital signatures, commitment generation, and audit requests are delegated to additional CPU cores or hosts. With multiple hosts, each host must build a replica of the history tree which can be done at least as fast as our maximum insert rate of 38,000 events per second. Additional CPU cores on these hosts can then be used for generating commitments or handling audit requests.

For some applications, 38,000 events per second may still not be fast enough. Scaling beyond this would require fragmenting the event insertion and storage tasks across multiple logs. To break interdependencies between them, the fundamental history tree data structure we presently use would need to evolve, perhaps into disjoint logs that occasionally entangle with one another as in timeline entanglement [43]. Designing and evaluating such a structure is future work.

6.2 Logs larger than RAM

For exceptionally large audits or queries, where the working set size does not fit into RAM, we observed that throughput was limited to disk seek latency. Similar issues occur in any database query system that uses secondary storage, and the same software and hardware techniques used by databases to speed up queries may be used, including faster or higher throughput storage systems or partitioning the data and storing it in-memory across a cluster of machines. A single large query can then be issued to the cluster node managing each sub-tree. The results would then be merged before transmitting the results to the auditor. Because each sub-tree would fit in its host's RAM, sub-queries would run quickly.

6.3 Signing batches of events

When large computer clusters are unavailable and the performance cost of DSA signatures is the limiting factor in the logger's throughput, we may improve performance of the logger by allowing multiple updates to be handled with one signature computation.

Normally, when a client requests an event X to be inserted, the logger assigns it an index i, generates the commitment C_i, signs it, and returns the result. If the logger has insufficient CPU to sign every commitment, the logger could instead delay returning C_i until it has a signature for some later commitment C_j ($j \geq i$). This later signed commitment could then be sent to the client expecting an earlier one. To ensure that the event X_i in the log committed by C_j was X, the client may request a membership proof from commitment C_j to event i and verify that $X_i = X$. This is safe due to the tamper-evidence of our structure. If the logger were ever to later sign a C_i inconsistent with C_j, it would fail an incremental proof.

In our prototype, inserting events into the log is twenty times faster than generating and signing commitments. The logger may amortize the costs of generating a signed commitment over many inserted events. The number of events per signed commitment could vary dynamically with the load on the logger. Under light load, the logger could sign every commitment and insert 1,750 events per second. With increasing load, the logger might sign one in every 16 commitments to obtain an estimated insert rate of 17,000 events per second. Clients will still receive signed commitments within a fraction of a second, but several clients can now receive the same commitment. Note that this analysis only considers the maximum insert rate for the log and does not include the costs of replying to audits. The overall performance improvements depend on how often clients request incremental and membership proofs.

7 Related work

There has been recent interest in creating append-only databases for regulatory compliance. These databases permit the ability to access old versions and trace tampering [51]. A variety of different data structures are used, including a B-tree [64] and a full text index [47]. The security of these systems depends on a write-once semantics of the underlying storage that cannot be independently verified by a remote auditor.

Forward-secure digital signature schemes [3] or stream authentication [21] can be used for signing commitments in our scheme or any other logging scheme. Entries in the log may be encrypted by clients for privacy. Kelsey and Schneier [57] have the logger encrypt entries with a key destroyed after use, preventing an attacker from reading past log entries. A hash function is iterated to generate the encryption keys. The initial hash is sent to a trusted auditor so that it may decrypt events. Logcrypt [29] extends this to public key cryptography.

Ma and Tsudik [41] consider tamper-evident logs built using forward-secure sequential aggregating signature schemes [39, 40]. Their design is round-based. Within each round, the logger evolves its signature, combining a new event with the existing signature to generate a new signature, and also evolves the authentication key. At the end of a round, the final signature can authenticate any event inserted.

Davis et. al. [17] permits keyword searching in a log by trusting the logger to build parallel hash chains for each keyword. Techniques have also been designed for keyword searching encrypted logs [60, 61]. A tamper-evident store for voting machines has been proposed, based on append-only signatures [33], but the signature sizes grow with the number of signed messages [6].

Many timestamping services have been proposed in the literature. Haber and Stornetta [27] introduce a time-stamping service based on hash chains, which influenced the design of Surety, a commercial timestamping service that publishes their head commitment in a newspaper once a week. Chronos is a digital timestamping service inspired by a skip list, but with a hashing structure similar to our history tree [7]. This and other timestamping designs [9, 10] are round-based. In each round, the logger collects a set of events and stores the events within that round in a tree, skip list, or DAG. At the end of the round the logger publicly broadcasts (e.g., in a newspaper) the commitment for that round. Clients then obtain a logarithmically-sized, tamper-evident proof that their events are stored within that round and are consistent with the published commitment. Efficient algorithms have been constructed for outputting time stamp authentication information for successive events within a round in a streaming fashion, with minimal storage on the server [37]. Unlike these systems, our history tree allows events to be added to the log, commitments generated, and audits to be performed at any time.

Maniatis and Baker [43] introduced the idea of *timeline entanglement*, where every participant in a distributed system maintains a log. Every time a message is received, it is added to the log, and every message transmitted contains the hash of the log head. This process spreads commitments throughout the network, making it harder for malicious nodes to diverge from the canonical timeline without there being evidence somewhere that could be used in an audit to detect tampering. Auditorium [55] uses this property to create a shared "bulletin board" that can detect tampering even when $N-1$ systems are faulty.

Secure aggregation has been investigated as a distributed protocol in sensor networks for computing sums, medians, and other aggregate values when the host doing the aggregation is not trusted. Techniques include trading off approximate results in return for sublinear communication complexity [12], or using MAC codes to detect one-hop errors in computing aggregates [30]. Other aggregation protocols have been based around hash tree structures similar to the ones we developed for Merkle aggregation. These structures combine aggregation and cryptographic hashing, and include distributed sensor-network aggregation protocols for computing authenticated sums [13] and generic aggregation [45]. The sensor network aggregation protocols interactively generate a secure aggregate of a set of measurements. In Merkle aggregation, we use intermediate aggregates as a tool for performing efficient queries. Also, our Merkle aggregation construction is more efficient than these designs, requiring fewer cryptographic hashes to verify an event.

8 Conclusions

In this work we have shown that regular and continous auditing is a critical operation for any tamper-evident log system, for without auditing, clients cannot detect if a Byzantine logger is misbehaving by not logging events, removing unaudited events, or forking the log. From this requirement we have developed a new tamper-evident log design, based on a new Merkle tree data structure that permits a logger to produce concise proofs of its correct behavior. Our system eliminates any need to trust the logger, instead allowing clients and auditors of the logger to efficiently verify its correct behavior with only a constant amount of local state. By sharing commitments among clients and auditors, our design is resistant even to sophisticated forking or rollback attacks, even in cases where a client might change its mind and try to repudiate events that it had logged earlier.

We also proposed Merkle aggregation, a flexible mechanism for encoding auxiliary attributes into a Merkle tree that allows these attributes to be aggregated from the leaves up to the root of the tree in a verifiable fashion. This technique permits a wide range of efficient, tamper-evident queries, as well as enabling verifiable, safe deletion of "expired" events from the log.

Our prototype implementation supports thousands of events per second, and can easily scale to very large logs. We also demonstrated the effectiveness of Bloom filters to enable a broad range of queries. By virtue of its concise proofs and scalable design, our techniques can be applied in a variety of domains where high volumes of logged events might otherwise preclude the use of tamper-evident logs.

Acknowledgements

The authors gratefully acknowledge Farinaz Koushanfar, Daniel Sandler, and Moshe Vardi for many helpful comments and discussions on this project. The authors also thank the anonymous referees and Micah Sherr, our shepherd, for their assistance. This work was supported, in part, by NSF grants CNS-0524211 and CNS-0509297.

References

[1] ACCORSI, R., AND HOHL, A. Delegating secure logging in pervasive computing systems. In *Security in Pervasive Computing* (York, UK, Apr. 2006), pp. 58–72.

[2] ANAGNOSTOPOULOS, A., GOODRICH, M. T., AND TAMASSIA, R. Persistent authenticated dictionaries and their applications. In *International Conference on*

Information Security (ISC) (Seoul, Korea, Dec. 2001), pp. 379–393.

[3] BELLARE, M., AND MINER, S. K. A forward-secure digital signature scheme. In *CRYPTO '99* (Santa Barbara, CA, Aug. 1999), pp. 431–448.

[4] BELLARE, M., AND YEE, B. S. Forward integrity for secure audit logs. Tech. rep., University of California at San Diego, Nov. 1997.

[5] BENALOH, J., AND DE MARE, M. One-way accumulators: a decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '93)* (Lofthus, Norway, May 1993), pp. 274–285.

[6] BETHENCOURT, J., BONEH, D., AND WATERS, B. Cryptographic methods for storing ballots on a voting machine. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2007).

[7] BLIBECH, K., AND GABILLON, A. CHRONOS: An authenticated dictionary based on skip lists for timestamping systems. In *Workshop on Secure Web Services* (Fairfax, VA, Nov. 2005), pp. 84–90.

[8] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[9] BULDAS, A., LAUD, P., LIPMAA, H., AND WILLEMSON, J. Time-stamping with binary linking schemes. In *CRYPTO '98* (Santa Barbara, CA, Aug. 1998), pp. 486–501.

[10] BULDAS, A., LIPMAA, H., AND SCHOENMAKERS, B. Optimally efficient accountable time-stamping. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)* (Melbourne, Victoria, Australia, Jan. 2000), pp. 293–305.

[11] CAMENISCH, J., AND LYSYANSKAYA, A. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO '02* (Santa Barbara, CA, Aug. 2002), pp. 61–76.

[12] CHAN, H., PERRIG, A., PRZYDATEK, B., AND SONG, D. SIA: Secure information aggregation in sensor networks. *Journal Computer Security 15*, 1 (2007), 69–102.

[13] CHAN, H., PERRIG, A., AND SONG, D. Secure hierarchical in-network aggregation in sensor networks. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, VA, Oct. 2006), pp. 278–287.

[14] CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. Private information retrieval. In *Annual Symposium on Foundations of Computer Science* (Milwaukee, WI, Oct. 1995), pp. 41–50.

[15] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested append-only memory: Making adversaries stick to their word. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 189–204.

[16] D. S. PARKER, J., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering 9*, 3 (1983), 240–247.

[17] DAVIS, D., MONROSE, F., AND REITER, M. K. Time-scoped searching of encrypted audit logs. In *Information and Communications Security Conference* (Malaga, Spain, Oct. 2004), pp. 532–545.

[18] DEUTSCH, P. Gzip file format specification version 4.3. RFC 1952, May 1996. http://www.ietf.org/rfc/rfc1952.txt.

[19] DEVANBU, P., GERTZ, M., KWONG, A., MARTEL, C., NUCKOLLS, G., AND STUBBLEBINE, S. G. Flexible authentication of XML documents. *Journal of Computer Security 12*, 6 (2004), 841–864.

[20] DEVANBU, P., GERTZ, M., MARTEL, C., AND STUBBLEBINE, S. G. Authentic data publication over the internet. *Journal Computer Security 11*, 3 (2003), 291–314.

[21] GENNARO, R., AND ROHATGI, P. How to sign digital streams. In *CRYPTO '97* (Santa Barbara, CA, Aug. 1997), pp. 180–197.

[22] GERR, P. A., BABINEAU, B., AND GORDON, P. C. Compliance: The effect on information management and the storage industry. The Enterprise Storage Group, May 2003. http://searchstorage.techtarget.com/tip/0,289483, sid5_gci906152,00.html.

[23] GOH, E.-J. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/ See also http://eujingoh.com/papers/secureindex/.

[24] GOODRICH, M., TAMASSIA, R., AND SCHWERIN, A. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II (DISCEX II)* (Anaheim, CA, June 2001), pp. 68–82.

[25] GOODRICH, M. T., TAMASSIA, R., TRIANDOPOULOS, N., AND COHEN, R. F. Authenticated data structures for graph and geometric searching. In *Topics in Cryptology, The Cryptographers' Track at the RSA Conference (CT-RSA)* (San Francisco, CA, Apr. 2003), pp. 295–313.

[26] GOYAL, V., PANDEY, O., SAHAI, A., AND WATERS, B. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security (CCS '06)* (Alexandria, Virginia, Oct. 2006), pp. 89–98.

[27] HABER, S., AND STORNETTA, W. S. How to time-stamp a digital document. In *CRYPTO '98* (Santa Barbara, CA, 1990), pp. 437–455.

[28] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *SOSP '07* (Stevenson, WA, Oct. 2007).

[29] HOLT, J. E. Logcrypt: Forward security and public verification for secure audit logs. In *Australasian Workshops on Grid Computing and E-research* (Hobart, Tasmania, Australia, 2006).

[30] HU, L., AND EVANS, D. Secure aggregation for wireless networks. In *Symposium on Applications and the Internet Workshops (SAINT)* (Orlando, FL, July 2003), p. 384.

[31] ITKIS, G. Cryptographic tamper evidence. In *ACM Conference on Computer and Communications Security (CCS '03)* (Washington D.C., Oct. 2003), pp. 355–364.

[32] KELSEY, J., CALLAS, J., AND CLEMM, A. Signed Syslog messages. http://tools.ietf.org/id/draft-ietf-syslog-sign-23.txt (work in progress), Sept. 2007.

[33] KILTZ, E., MITYAGIN, A., PANJWANI, S., AND RAGHAVAN, B. Append-only signatures. In *International Colloquium on Automata, Languages and Programming* (Lisboa, Portugal, July 2005).

[34] KOCHER, P. C. On certificate revocation and validation. In *International Conference on Financial Cryptography*

(FC '98) (Anguilla, British West Indies, Feb. 1998), pp. 172–177.

[35] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *SOSP '07* (Stevenson, WA, Oct. 2007), pp. 45–58.

[36] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Operating Systems Design & Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[37] LIPMAA, H. On optimal hash tree traversal for interval time-stamping. In *Proceedings of the 5th International Conference on Information Security (ISC02)* (Seoul, Korea, Nov. 2002), pp. 357–371.

[38] LONVICK, C. The BSD Syslog protocol. RFC 3164, Aug. 2001. http://www.ietf.org/rfc/rfc3164.txt.

[39] MA, D. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS'08)* (Tokyo, Japan, Mar. 2008), pp. 341–352.

[40] MA, D., AND TSUDIK, G. Forward-secure sequential aggregate authentication. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Oakland, CA, May 2007), IEEE Computer Society, pp. 86–91.

[41] MA, D., AND TSUDIK, G. A new approach to secure logging. *Transactions on Storage 5*, 1 (2009), 1–21.

[42] MANIATIS, P., AND BAKER, M. Enabling the archival storage of signed documents. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Monterey, CA, 2002).

[43] MANIATIS, P., AND BAKER, M. Secure history preservation through timeline entanglement. In *USENIX Security Symposium* (San Francisco, CA, Aug. 2002).

[44] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[45] MANULIS, M., AND SCHWENK, J. Provably secure framework for information aggregation in sensor networks. In *Computational Science and Its Applications (ICCSA)* (Kuala Lumpur, Malaysia, Aug. 2007), pp. 603–621.

[46] MERKLE, R. C. A digital signature based on a conventional encryption function. In *CRYPTO '88* (1988), pp. 369–378.

[47] MITRA, S., HSU, W. W., AND WINSLETT, M. Trustworthy keyword search for regulatory-compliant records retention. In *International Conference on Very Large Databases (VLDB)* (Seoul, Korea, Sept. 2006), pp. 1001–1012.

[48] MONTEIRO, S. D. S., AND ERBACHER, R. F. Exemplifying attack identification and analysis in a novel forensically viable Syslog model. In *Workshop on Systematic Approaches to Digital Forensic Engineering* (Oakland, CA, May 2008), pp. 57–68.

[49] NAOR, M., AND NISSIM, K. Certificate revocation and certificate update. In *USENIX Security Symposium* (San Antonio, TX, Jan. 1998).

[50] OSTROVSKY, R., SAHAI, A., AND WATERS, B. Attribute-based encryption with non-monotonic access structures. In *ACM Conference on Computer and Communications Security (CCS '07)* (Alexandria, VA, Oct. 2007), pp. 195–203.

[51] PAVLOU, K., AND SNODGRASS, R. T. Forensic analysis of database tampering. In *ACM SIGMOD International Conference on Management of Data* (Chicago, IL, June 2006), pp. 109–120.

[52] PETERSON, Z. N. J., BURNS, R., ATENIESE, G., AND BONO, S. Design and implementation of verifiable audit trails for a versioning file system. In *USENIX Conference on File and Storage Technologies* (San Jose, CA, Feb. 2007).

[53] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.

[54] SAHAI, A., AND WATERS, B. Fuzzy identity based encryption. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt '05)* (May 2005), vol. 3494, pp. 457 – 473.

[55] SANDLER, D., AND WALLACH, D. S. Casting votes in the Auditorium. In *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)* (Boston, MA, Aug. 2007).

[56] SCHNEIER, B., AND KELSEY, J. Automatic event-stream notarization using digital signatures. In *Security Protocols Workshop* (Cambridge, UK, Apr. 1996), pp. 155–169.

[57] SCHNEIER, B., AND KELSEY, J. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security 1*, 3 (1999).

[58] SION, R. Strong WORM. In *International Conference on Distributed Computing Systems* (Beijing, China, May 2008), pp. 69–76.

[59] SNODGRASS, R. T., YAO, S. S., AND COLLBERG, C. Tamper detection in audit logs. In *Conference on Very Large Data Bases (VLDB)* (Toronto, Canada, Aug. 2004), pp. 504–515.

[60] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy* (Berkeley, CA, May 2000), pp. 44–55.

[61] WATERS, B. R., BALFANZ, D., DURFEE, G., AND SMETTERS, D. K. Building an encrypted and searchable audit log. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2004).

[62] WEATHERSPOON, H., WELLS, C., AND KUBIATOWICZ, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Future Directions in Distributed Computing* (2003), vol. 2584 of *Lecture Notes in Computer Science*, pp. 142–147.

[63] YUMEREFENDI, A. R., AND CHASE, J. S. Strong accountability for network storage. *ACM Transactions on Storage 3*, 3 (2007).

[64] ZHU, Q., AND HSU, W. W. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *ACM SIGMOD International Conference on Management of Data* (Baltimore, MD, June 2005), pp. 395–406.